GSWO: A Programming Model for GPU-enabled Parallelization of Sliding Window Operations in Image Processing

***Abstract*—**Sliding Window Operations (SWOs) are widely used in image processing applications. They often have to be performed repeatedly across the target image, which can demand significant computing resources when processing large images with large windows. In applications in which real-time performance is essential, running these filters on a CPU often fails to deliver results within an acceptable timeframe. The emergence of sophisticated graphic processing units (GPUs) presents an opportunity to address this challenge. However, GPU programming requires a steep learning curve and is error-prone for novices, so the availability of a tool that can produce a GPU implementation automatically from the original CPU source code can provide an attractive means by which the GPU power can be harnessed effectively. This paper presents a GPU-enabled programming model, called GSWO, which can assist GPU novices by converting their SWO-based image processing applications from the original C/C++ source code to CUDA code in a highly automated manner. This model includes a new set of simple SWO pragmas to generate GPU kernels and to support effective GPU memory management. We have implemented this programming model based on a CPU-to-GPU translator (C2GPU). Evaluations have been performed on a number of typical SWO image filters and applications. The experimental results show that the GSWO model is capable of efficiently accelerating these applications, with improved applicability and a speed-up of performance compared to several leading CPU-to-GPU source-to-source translators.

Index Terms— Parallel Computing, Sliding Window Operation, CUDA, Automatic Translation

I. Introduction

Sliding Window Operations (SWOs) are performed very frequently in image processing and analysis [1-3]. A typical SWO repeatedly applies an image filter to a pre-defined sub-window that slides progressively across the target image. Many filters involve logical and mathematical operations with high complexity. Examples include *rank-order filters* (adaptive two-pass filter [4], fuzzy rank LUM filter [5], etc.), which involve sorting values of the pixels in the window in ascending order; and *morphological filters* (directional morphological filter [6], dilation and erosion filter [7], etc.), which perform morphological operations such as erosion, dilation, opening and closing by using a moving window. Since SWOs need to be performed repeatedly across the entire target image (see Figure 1), significant computing resources are required when processing large images with large windows. If real-time performance is essential, running these filters on a CPU may not deliver the results within an acceptable time.

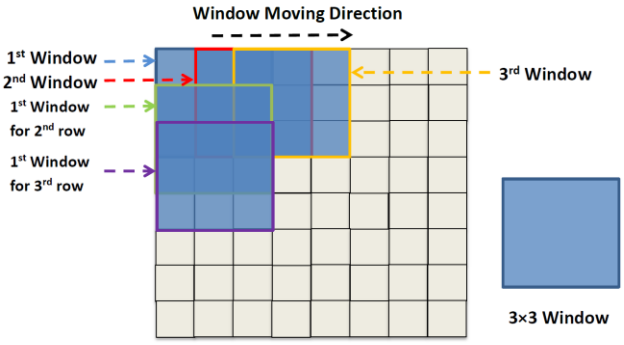


Figure 1. Illustration of a Sliding Window

Faced by these computational demands, many researchers have adopted parallel computing [8-11]. As thousands of computing threads are available nowadays on individual graphics cards, graphics processing units (GPUs) have become increasingly popular for handling computationally intensive tasks that can be parallelized [12]. SWOs are particularly suitable for this as the calculations associated with different window locations are independent of each other and can thus be executed in parallel.

Unfortunately, the parallelization of CPU code for execution on GPUs is not straightforward, and manual implementation typically involves significant effort. Indeed, designing and implementing GPU algorithms that utilize the GPU potential most effectively requires an in-depth knowledge of the underlying GPU architecture. In this process, non-experts are error prone, and novices experience a steep learning curve.

Researchers/programmers already have access to a number of GPU-enabled image processing libraries or open-source code, such as OpenCV_GPU for CUDA [13], GPUCV [14], ethothepi-CUDA-Image-Processing [15], CUDA/NPP library (for morphological operations) [8], CUDA-based denoising filters [16], etc., but we have observed that the GPU-enabled filters from these existing resources do not meet a wide range of user demands. Although the filters have been carefully conceived by experts for optimal performance gain on GPUs, in general each is designed for a specific purpose and has little extendibility – no customized development is allowed. Further, since overall they cover only a limited set of standard SWOs, often researchers/programmers still have to write their own SWO code either to modify the standard filters or to implement

new SWOs. The difficulties inherent in GPGPU algorithm design and implementation make it highly desirable to have a means by which users can write the GPGPU code they require even if they lack an in-depth knowledge of GPUs.

This paper provides a proof of concept demonstrating the possibility of employing an easy-to-use CPU-to-GPU code translator to accelerate SWO-based image filter applications to deliver markedly improved performance over CPU-based approaches. It presents a new programming model, *GSWO*, that allows GPU-enabled parallelization of SWOs for image processing in a highly automated manner – the users can annotate their source code for image filters using pragmas, and the annotated code is then automatically converted into GPU code with optimized parameter settings. GSWO is based on a web-based platform, *C2GPU*, which supports automated code conversion from CPU to GPU [17].

Our implementation of the GSWO model produced a set of newly defined pragmas that support CPU-to-GPU conversion of a variety of SWO source codes. By using these pragmas, users can generate GPGPU code for their SWO image filters without having a good knowledge of GPUs, thus supporting customized development in many image applications and affording the possibility of remarkable performance gains.

In summary, the main contributions of the paper are:

- An annotation-based programming model, GSWO, is presented and implemented for automated CPU-to-GPU translation of SWOs for image processing. The model features new SWO pragmas that are easy to use and are applicable to many types of parallelizable operations in sliding windows. It also introduces a memory management hierarchy for effective memory creation and data transfer between CPU and GPU.

- A thorough performance evaluation of the GSWO model using benchmarks and practical applications has been carried out, the results of which suggest notable performance gains and improved usability for SWO filter applications compared to other leading CPU-to-GPU translators [23-30].

The rest of the paper is organized as follows. Section 2 provides a brief overview of related work, and we present the proposed GSWO programming model in Section 3 and the experimental validation results in Section 4. Section 5 draws conclusions from the work and suggests areas for future investigation.

II. Related Work

This section gives a brief introduction to SWOs and a survey of the existing CPU-to-GPU source-to-source translators.

A. *SWOs for image filtering*

In a typical image filter, a user-defined window moves in a raster-scan order until the entire image is covered as shown in Figure 1. We denote an $N \times M$-sized image by $D_{n,m}$: $n = 1,..,N$; $m = 1,..,M$; an $I \times J$-sized sliding window by $W_{i,j}$: $i = 1,..,I$; $j = 1,..,J$; and the set of r-step operations within this sliding window by $P(p_{1,..,T})$. Table 1 shows a typical implementation of an SWO on a CPU.

Table 1. Work flow of an SWO in image processing

```
1:   // Initialisation and set memory.
2:    float D[N][M] = ReadInputImage();
3:    int start_point_x = 0 ;
4:    int start_point_y = 0 ;
5:
6:   // Outer Loop start for whole image D_{n,m}
7:    for n = 1: N
8:    for m = 1: M
9:        start_point_x = n ;
10:       start_point_y = m ;
11:       float W[I][J] = CopyDataFromInputImage(n, m, I, J);
12:
13:  // Nested Loop start for Sliding Window W_{i,j}
14:       for i = 1: I
15:       for j = 1: J
16:
17:  // Execute operations P(p_{1,..,T})
18:        p1; p2; p3;……; pT
19:       end; end;
20:    CopyDataToInputImage (D[N][M], W[I][J]);
21:    end; end;
22:
```

If the execution time of operations $P(p_{1,..,T})$ is assumed to be t, the running time of the SWO in Table 1 is:

$$T = N \times M \times I \times J \times t \qquad (1)$$

For operations with high complexity (i.e. large t) that use large images and windows (i.e. large N, M, I, J), Equation (1) shows that the SWO can become very time consuming.

B. *Computationally intensive image filters*

SWOs are typically used in image filters, many of which are computationally intensive; rank-order filters [4-5] and morphological filters [6-8] are two typical examples.

Rank-order filters are generally used for noise removal [4] and often involve sorting the values of the pixels in the sliding window into ascending order, which is time-consuming. Well known rank-order filters include low-upper-middle filter [18], mode filter [19], alpha-trimmed mean filter [20] and median filter [9]. Much attention has been paid to accelerating these filters in image processing.

Morphological filters are widely used to extract edges or skeletons of images in applications such as remote sensing image recognition [11] and document image analysis [21]. The computational cost of morphological filters mainly comes from recursive erosion, dilation, opening and closing transforms [22]. Increasing the size of the structuring elements can add a significant extra computational cost to the filters [8].

Rank-order filters and morphological filters are only two examples of uses of SWOs in image processing. Many other image filters involve a wide variety of SWOs, most of which cannot be represented by standard filters.

Table 2. Comparison of properties of typical directive-based tools

	hiCUDA [30]	PGI (OpenACC) [32]	MINT [28]	CUDA-lite [29]
Language support	C-to-CUDA	C++/Fortan-to-CUDA	C-to-CUDA	CUDA-to-CUDA
Easy-use of directives	Complex	Complex	Easy	Easy
Applicability	Good	Outstanding	Limited	Good
Speedup performance	Good	Good	Outstanding	Good
Optimisation option	Use of shared memory	No particular one	Shared memory and loop aggregation	Improved memory hierarchy
Readability of GPU code	Moderate	No	Good	Good

With this in mind, the GSWO model and its implementation (i.e. pragmas) have been designed to be capable of performing CPU-to-GPU source conversion from arbitrary SWO code in a highly automated manner. This will be particularly valuable to researchers who are committed to the implementation of non-standard, compute-intensive image filters in an innovative application, but who lack basic GPU skills.

C. *Existing CPU-to-GPU source translators*

Existing CPU-to-GPU source translators can be classified into three categories [23], based on algorithmic skeletons [24], polyhedral models [25-27] and directives [28-32], respectively.

Algorithmic skeleton based tools adopt the idea of generating efficient target code by specific algorithm classes, such as SkePU [24]. Advantageously, they have highly optimized library implementations for each algorithm class. However, algorithmic skeleton tools demand that users manually implement and add a new algorithm skeleton if one is not available for a specific class of CPU code. Also, their usability is often low due to the difficulties involved in rewriting the original CPU source code and in defining algorithm classes and their corresponding skeletons.

Polyhedral model based tools translate source code with affine loop structures by performing dependency analysis and loop transformation (Par4All [27], Pluto [40]). While they require little input from the users, they are applicable only to source code with affine loop structures. This means that the polyhedral model can deal only with loop nests with affine bounds and conditional expressions.

Thus, while these two approaches can efficiently cope with the automatic parallelization of some known algorithm templates and certain types of loops, they are both highly sensitive to the characteristics and data structures of the input CPU source code. In image filter applications using SWOs, this implies that any new image filter has to be manually implemented and added as an extra class. A further issue associated with these two approaches is the highly laborious task of identifying parallelizable regions and revising the relevant code. This drawback significantly limits their wide acceptance by programmers. Because of these limitations, the two approaches above are not considered in this paper.

Directive-based CPU-to-GPU source translation tools [28-32] offer a semi-automatic way of generating GPU code. They can generate GPU source code by manually adding annotations to the input CPU source code. Since users can directly insert annotations into their own code, the range of applications such translators support is much wider than those supported by algorithmic skeleton or polyhedral model based tools.

We collected a number of typical directive based translators and have compared their performance in Table 2. This indicates that most of the directive-based tools can process only C, and not C++, which is a significant disadvantage for use in application areas such as image processing.

The commercial compiler PGI accelerator [32] accelerates applications written in C++ by adding OpenACC [31] directives, but its pragmas are far too complex, and the GPGPU code it outputs is almost unreadable (since PGI is designed as a compiler instead of a source-to-source translator).

CUDA-lite [29] introduces directives to improve the memory hierarchy of CUDA by directly inserting the directives into the CUDA code. However, it is not a CPU-to-GPU source-to-source conversion tool.

hiCUDA [30] provides programmers with a set of pragmas mapping to typical CUDA operations. The CUDA codes in hiCUDA are optimized by dealing with global memory and transformations to leverage the complex memory hierarchy. A weakness is that hiCUDA requires users to have sufficient GPU knowledge to be able to specify the threads and thread blocks.

MINT [28] is a very easy-to-use C-to-CUDA source translator containing five types of pragma. It is designed for accelerating stencil computations on NVIDIA GPUs only. This translator accepts C source input with some intuitive MINT directives to generate highly optimized CUDA C which may produce a performance gain of up to 10×.

D. *Limitations of MINT*

Simplicity is a major goal of the directive design in MINT, and it incorporates several easy-to-use pragmas: ***parallel***, ***for***, ***copy*** and ***single***. While these pragmas are sufficient to deal with simple C code, they cannot support SWOs in image processing due to following limitations.

- The ***copy*** pragma in MINT combines memory allocation and data transfer. However, SWOs in image processing need to separate these two operations in order to allow the reuse of the allocated memory for data transfer (which may occur many times between the CPU and GPU), without having to involve memory re-allocation each time.
- Only stencil computing is supported in the kernel generation; SWOs in image processing employ many operations $S(p_{1,\ldots,T})$ that are not supported by MINT.
- The pragma ***parallel*** must be located immediately behind the pragma ***copy***, which means that MINT cannot handle algorithms in which we need to insert source code between these two pragmas.

The work presented in this paper is directive-based in order to meet the demands of flexibility and extendibility that image processing presents. Users are able to annotate their code using the proposed GSWO pragmas to achieve parallelization of a variety of SWOs in image processing. The GSWO pragmas represent significant improvements over the pragmas in MINT as they support a wide variety of SWOs for image processing. The SWO model also features effective memory management, allowing for superior performance over the majority of existing CPU-to-GPU translators.

III. GSWO Programming Model

GSWO performs CPU-to-GPU source conversion and was developed as part of the C2GPU toolkit [17], the system architecture of which is based on that of MINT [28], but with a number of extended components. More details of the C2GPU toolkit can be found in [17] and in Figure 9. GSWO follows the system design in MINT [28], which comprises a host processor and an accelerator, and is neutral about all of the data transfers between them. To accelerate SWO-based image filter applications, each thread deals only with the operations within a single sliding window. The GPU parallelization of SWOs also implies the following assumptions: no parallelization within the sliding window, and no input data reuse between the sliding windows. Table 3 shows a simple example presenting the code of a 3×3 median filter implementation using a GSWO model. A list of the GSWO pragmas is given in Table 5.

Table 3. Example of a 3×3 SWO-based median filter

CPU Code

```
1:  #pragma parallel {
2:  ……..
3:  #pragma single initialization {
4:          float v[9] = {0,0,0,0,0,0,0,0,0};          }
5:
6:  #pragma for  nest(2) tile(16,16)
7:          for ( i = 1; i <= height ; i++)
8:          for ( j = 1; j <= width ; j++) {
9:
10:  #pragma single transfer {
11:          v[0] = Image [i-1][j-1] ;
12:          v[1] = Image [i-1][j] ;
13:          ……..
14:          v[8] = Image [i+1][j+1];                  }
15:
16: #pragma single remain {
17:          for (m = 0 ; m < 9 ; m++)
18:          for (t = m+1; t < 9; t++) {
19:              if(v[m] > v[t]) {
20:                  tmp = v[m];
21:                  v[m] = v[t];
22:                  v[t] = tmp ;  }}                  }
23 :                                                  }
24: #pragma single assign {
25:     Image[i][j] = v[4] ;                          }
26:                                  }
27:                 }
```

GPU parallelization of the SWOs median filter in Table 3 begins with using "*#pragma parallel*" to point out the parallelizable region of CPU code. Within this region, the pragma of "*single initialisation*" defines a one dimensional float array for storing the pixel information in a 3×3 sliding window. Then we use "*#pragma for nest(2) tile(16,16)"* to mark the nested loops for GPU acceleration. The clauses "*nest*" and "*tile*" inherited from MINT [28] respectively indicate the depth of for-loop and specify how the iteration space of a loop nest is to be subdivided into "*tiles*".

Inside the nested loops, the pragma of "*transfer*" covers the CPU code of transferring the 2D image data of a 3×3 sliding window into a 1D float array, which had been marked by the pragma "*initialization*". The pragma "*remain*" includes the CPU code to obtain the median from this 1D float array. Lastly, the pragma "*assign*" marks the CPU code to transfer the new data from the1D float array to the corresponding 2D image data associated with a 3×3 sliding window. In the GSWO model, the CPU code highlighted by such pragmas will be automatically translated into GPU code. The implementation of the GSWO model is discussed in detail below.

A. *Parallelization of SWOs*

As seen in Table 1, the algorithm structure of an SWO in image processing is very suitable for parallelization. Its GPU-enabled implementation typically follows the steps below (see Table 4 for a detailed example of an implementation):

1. A GPU device memory buffer D^{GPU} is created and allocated to store the complete image data, which is transferred from the CPU host buffer $D_{n,m}$ to the GPU device buffer D^{GPU} .
2. The GPU kernel parameters are registered; the number blocks and threads are determined.
3. The operations $P(p_{1,..,T})$ are rewritten in the kernel as individual functions; the GPU kernel function is called after registering the GPU kernel parameters.
4. The processed image data in D^{GPU} are transferred back to the host buffer $D_{n,m}$.

Table 4. A GPU implementation of SWO in image processing

```
1:  {        // Initialisation and create GPU memory.
2:           CreateGPUMemory(D^GPU)
3:          // Transfer whole image data from CPU to GPU
4:           TransferDataFromCPUtoGPU(D_n,m , D^GPU )
5:          // Register GPU Kernel Parameters
6:           dim3 threads() ;
7:           dim3 blocks() ;
8:          // Calling GPU Kernel Function
9:           GSWO_Function<<<<blocks, threads>>>>( D^GPU, I, J, N, M);
10:         // Transfer whole image data from GPU to CPU
11:          TransferDataFromGPUtoCPU(D_n,m , D^GPU)
12: }
13:
14:    // Rewrite operations S(p_1,..,T) in Kernel Function
15: __global___ void GSWO_Function(D^GPU, I, J, N, M ){
16:         // Thread Index Calculation
17:         int _idy = blockIdx.y * blockDim.y + threadIdx.y ;
18:         int _idx = blockIdx.x * blockDim.x + threadIdx.x ;
19:         // Define variable to store SW data
20:         float  W^GPU = CopySWDataTo(D^GPU, I, J)
21:         // Execute operations S(p_1,..,T) in Kernel
22:             p_1; p_2; p_3;……; p_T
23:          // Transfer SW data from  W^GPU to D^GPU
24:          D^GPU = CopySWDataFrom(W^GPU, I, J)
25:  }
```

Table 5. Listing of GSWO model pragmas

	Directives	*Descriptions*
Basic pragma	**Parallel**	To identify a region generating a kernel function
	Parallel region	To identify a parallel region containing parallel work
	For	To mark the succeeding "**For**" loop for GPU acceleration
	Single	To indicate serial regions in the GPUSWO model
Memory Management	**CopyByTexture**	To create a CUDA texture on a device, and bind or unbind with 2D data
	CopyMalloc1DArray	To create a CUDA array on a device, associating it with a CUDA texture on the device
	CopyMemcopy2D	To create a CUDA *cudamemcpy2D* function to copy a matrix between CPU and GPU memory
	CopyMemcopy2DToArray	To create a CUDA function *cudaMemcpy2DToArray* to copy data between CPU and GPU memory
	CopyBindTexture	To bind the created texture memory to a CUDA global array
	Copy2DArrayTo1DArray	To convert the array with different dimensions on the CPU memory buffer
Kernel Generation	**Initialisation**	To define a one dimensional array for storing the data in a sliding window
	Transfer	To transform the code of putting the data in a sliding window into a local variable within a "**For**" loop
	Remain	To transform the operations on a sliding window from CPU algorithm to the GPU kernel.
	Assign	To assign the new data to the relevant GPU buffer with the correct index.
	Transfer	To transform the code of putting the data in a sliding window into a local variable within a "**For**" loop
Thread and Block Size (inherited from MINT [28])	**Nest ()**	To indicate the depth of for-loop parallelization within a loop nest
	Tile (t_x, t_y, t_z)	To specify how the iteration space of a loop nest is to be subdivided into tiles
	Chunksize (c_x, c_y, c_z)	To aggregate logical threads into a single CUDA thread.

The main acceleration should come from the parallelization of $S(p_{1,..,T})$ in each sliding window. In theory, the total running time of Equation (1) will be reduced from $N \times M \times I \times J \times O(t)$ to $I \times J \times O(t)$. However, in practice, overhead costs need to be considered, for example, the variables used to store the sliding window data have to be created and the data have to be transferred to the GPU.

B. *GSWO Model*

To allow for parallelization, three major types of directive are normally required.

- ***Identification of parallel region and kernel region***: These directives indicate parallel regions, which contain obviously parallel work, and regions generating GPU kernel code.
- ***Memory Management*:** These directives manage the tasks of memory allocation, conversion, transfer and optimization on the GPU and GPU buffers.
- ***Kernel Generation*:** These directives supervise the GPU kernel code generation.

The primary advantage of GSWO over MINT is that its memory management directives have an enhanced hierarchy. GSWO introduces a set of memory management pragmas to control GPU memory allocation, GPU-to-CPU memory transfer and CPU memory conversion, respectively. It also provides pragmas to allow for the use of texture memory (in addition to the use of global memory). These new pragmas bring the flexibility and effectiveness to memory management that is needed in SWOs for image processing.

In addition, GSWO introduces a set of newly defined kernel generation pragmas. These were designed by following the typical procedure of an SWO, which contains *initiation*, *transfer*, *remain* and *assign*. They are simple and can be applied to all types of parallelizable operations in sliding windows. Our experiments showed that using our pragmas provides a significant improvement in usability and productivity when compared with other CPU-to-GPU translators.

The final improvement of the GSWO model is that it extends the pragma ***parallel*** of MINT into two pragmas ***parallel*** *and* ***parallel region*** to distinguish the kernel region from the parallel region. The pragma ***parallel region*** indicates the start of a parallel region containing the CPU source code for parallelization, whereas the pragma ***parallel*** marks a loop for generating a GPU kernel function. This extension is similar to the directives ***parallel*** and ***kernels*** in the OpenACC standard, but it is less complicated, and easier to use by non-expert GPU programmers. With these two pragmas, the GSWO model can support a more complicated algorithm structure than MINT can.

C. *GSWO Pragmas*

A list of the GSWO pragmas is given in Table 4.

a) Basic pragmas

The basic pragma is similar to the pragma in MINT. The only difference is that:

- ***Parallel Region*** indicates the start of a region containing parallel work, and such regions within the block of this pragma will be accelerated***;***
- ***Parallel*** indicates the start of a region for kernel function generation, which normally contains "For" loops**.**

b) Memory management pragmas

For the memory management pragmas, GSWO extends the "**Copy**" pragma by allowing for memory allocation, data conversion and data transfer.

Two copy based pragmas are defined for memory allocation.

- ***CopyMalloc1DArray*** creates a CUDA array on the device, associating it with a CUDA texture memory.
- ***CopyByTexture*** creates CUDA texture memory on the device, binding (unbinding) it with 2D data (e.g. the image); this normally occurs in the initialization step.

Data transfer between the CPU and GPU includes two copy-based pragmas:

- ***CopyMemcopy2D*** is used to provide a CUDA function (*cudamemcpy2D*) to transfer a matrix in a normal data structure from the device memory to host memory.

- ***CopyMemcopy2DToArray*** creates a CUDA function *cudaMemcpy2DToArray* to copy data in a non-normal structure (Z-curve) between CPU and GPU memory.

A further pragma ***CopyBindTexture*** is defined to bind texture memory to a CUDA global array.

Data conversion is used to convert an array on the CPU memory buffer to different dimensions, for example, converting data from a 2D array to a 1D array for GPU use. One such pragma is ***Copy2DArrayTo1DArray***.

Examples of the memory management pragmas and their translations into CUDA code are illustrated in Table 9.

The data transfer and conversion pragmas used to transfer the data of the sliding windows are allocated within the ***parallel region*** pragma, but outside the ***parallel*** pragma. Also, the names of parameters in the memory creating pragmas correspond to the names of relevant 2D or 1D array variables.

c) Kernel generation pragmas

The kernel generation pragmas in GSWO are designed for GPU kernel code generation. They generate the kernel code to perform the SWOs and allow for correct data transfer between CPU and GPU. In the GSWO model, we have designed new "**single"** pragmas for kernel code generation, four of which are defined below.

- ***Single Initialisation***: generates CUDA kernel code that defines a 1D array with size *I*×*J* for storing the data in the sliding window. If the data in the sliding window are defined as a 2D array in CPU code, as seen in Figure 2, we need to define a 1D array to replace the 2D array outside of the "**For**" pragma in the CPU code. In translating to the CUDA kernel, the ***Single Initialisation*** pragma directly moves this statement into the kernel. In the CPU code, all of the referenced data can normally be defined in the "**For**" loop. However, in GSWO, all of the referenced data have to be defined outside the "**For**" loop (for the purpose of building the AST tree by ROSE).
- ***Single Transfer*** generates CUDA kernel code to transfer the data of the sliding window into the 1D array defined in the ***Single Initialisation*** pragma, so a 1D array with size 9 is used to store the data for a 3×3 sliding window. In our implementation, the CUDA kernel receives the data of the sliding window from the CUDA texture memory, as shown in Figure 2.
- ***Single Remain*** generates CUDA kernel code that corresponds to the operations on the sliding window. In our implementation, we simply copy the CPU source code to the CUDA kernel. By doing this, any user-written CPU source code can be converted into CUDA kernel code, as long as the target CPU source code is parallelizable, e.g. the variables are data independent between different loop iterations.
- ***Single Assign*** generates CUDA kernel code that copies the processed data in the sliding window to the relevant GPU buffer obtained via the thread and block IDs.

Figure 2 shows an example workflow of the kernel generation pragma in the GSWO programming model. Also, the CUDA kernel code generated from each kernel generation pragma is illustrated in Table 6, which represents sample code of a I×J window size image filter implementation on the CPU (left), and its converted CUDA code (right).

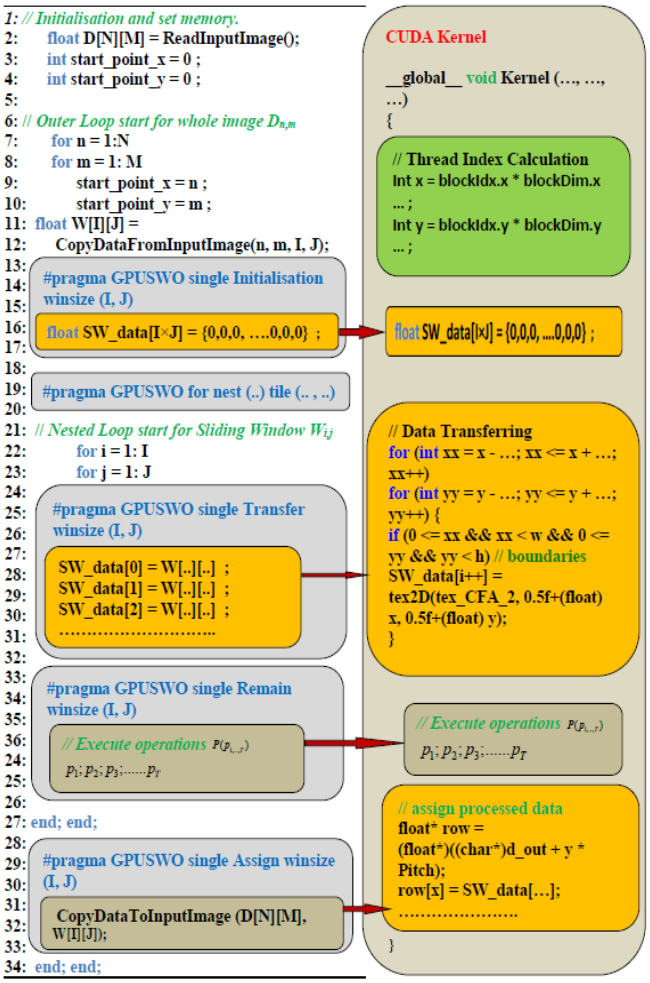


Figure 2. Work flow of the Kernel Generation Pragmas

D. *Block and Thread Size*

The selection of block and thread size in GSWO model is based on the pragmas in MINT: *nest*, *tile* and *chunksize*. As shown in Table 5, they are inherited and used by the GSWO model for indicating the depth of for-loop parallelization within a loop nest, specifying how the iteration space of a loop nest is to be subdivided into tiles, and aggregating logical threads into a single CUDA thread, respectively. The size of a CUDA thread block in the GSWO model is the same as in MINT: threads (t_x/c_x, t_y/c_y,t_z/c_z).

But the impact of selected block and thread size on acceleration in GSWO model is not as significant as that in MINT. The kernel generator in MINT makes all of the parameters in the function argument become kernel call parameters and makes all memory references through device memory. This requires code to be added into the kernel body to compute global thread IDs and references to be rewritten in terms of block and thread size. The mechanism of kernel generation in the GSWO model has been redefined as a simple way in Section III.C(c). The computation of global thread IDs is generated by default in the kernel body. The code for rewriting references is handled by each individual pragma.

Table 6. Benchmarks for evaluating the GSWO model

	Benchmarks	*Descriptions*
Rank-order filters	**MinFilter**	Get maximum value among all elements
	MaxFilter	Get minimum value among all elements
	MedianFilter	Get middle value after all elements are sorted numerically
	MidPointFilter (Mid-P)	Get an average value of maximum and minimum among all elements
	Alpha-Trimmed Mean Filter (Alpha-T)	Disregard the most atypical elements and calculate the mean value using those remaining
	Standard Deviation Filter (S-D)	Used to emphasize the local variability in an image
	Mode Filter	Replace pixels with the most frequently occurring pixel value selected from all elements
	Mean Filter	Find an average value among all elements
	Multi-stage directional median (M-D-M)	Used middle value obtained from the pixels set along four directions to edges
Morphological filters	**Erosion**	To shrink foreground elements and enlarge background elements with structure element
	Dilation	To enlarge foreground elements and shrink background elements with structure element
	Opening	To first do erosion and then do dilation with one structure element
	Closing	To first do dilation and then do erosion with one structure element
	Thinning	To do erosion and dilation with extended type of structure elements from hit and miss
	Thickening	To do erosion and dilation with extended type of structure elements from hit and miss
	Hit-and-miss	To do erosion and dilation with structure element introducing "do not care"
	Recursive erosion	To recursively do erosion operators with structure element
	Recursive dilation	To recursively do dilation operators with structure element
Practical Applications	**Camera Fingerprint Measurement**	IME company [36]
	Document Analysis	EU IMPACT project [37]

IV. Performance Evaluation

The GSWO model has been evaluated using a variety of use cases, including a set of SWO image filters and two image processing applications (camera fingerprint measurement and document segmentation [36, 37]). The cases selected are computationally expensive but parallelizable. The evaluation compares the computation time between the GPU and CPU. The baseline is the performance of the original CPU code on conventional hardware without the use of multi-threads; the evaluation tested the performance of the GSWO-generated CUDA, MINT-generated CUDA and OpenMP compared to this.

Table 7. CUDA code of Kernel Generation Pragmas

CPU Code

```
1:
2:  #pragma parallel {
3:  ……
4:  ……
3:  #pragma single initialisation{
4:  float v[9] = {0,0,0,0,0,0,0,0,0};
}
5:  #pragma for  nest(2)
    tile(16,16)
6: for ( i = 1; i <= height ; i++)
7:  for( j = 1; j <= width ; j++){
8:  #pragma single transfer{
9:      v[0] = Image [i-1][j-1] ;
10:     v[1] = Image [i-1][j] ;
11:     ……..
12:     v[8] = Image [i+1][j+1];  }
13:  #pragma single remain{
14:     for (m = 0 ; m < 9 ; m++)
15:       for (t = m+1; t < 9; t++) {

16:       if(v[m] > v[t]) {
17:             tmp = v[m];
18:             v[m] = v[t];
19:             v[t] = tmp ;  }} }

20:  #pragma single assign {
21:      Image[i][j] = v[4] ; }
         }
```

GPU Kernel

```
__global__ void kernel(int Pitch, float
*d_out,  int w, int h){
// index caculation
int x = blockIdx.x * blockDim.x +
threadIdx.x;
int y = blockIdx.y * blockDim.y +
threadIdx.y;
int i = 0;
float v[9] = {0,0,0,0,0,0,0,0,0};
// data transfer
for (int xx = x - 1; xx <= x + 1; xx++)
for (int yy = y - 1; yy <= y + 1; yy++) {
if (0 <= xx && xx < w && 0 <= yy &&
yy < h) // boundaries
v[i++] = tex2D(tex_CFA_2, 0.5f+(float)
x, 0.5f+(float) y);}
// directly copy from CPU code
 for (m = 0 ; m < 9 ; m++)
     for (t = m+1; t < 9; t++) {
         if(v[m] > v[t]) {
             tmp = v[m];
         v[m] = v[t];
             v[t] = tmp ;  }}
// pick the middle one
float* row = (float*)((char*)d_out + y *
Pitch);
row[x] = v[4];
}
```

The evaluation platforms were: (a) Intel Core i7-2670QM CPU and NVIDIA GeForce GT 540M; (b) Intel Core i7-3770K CPU and NVIDIA GeForce GTX 690; (c) Intel Core i7-2700K CPU and NVIDIA GeForce GTX 680. All used NVIDIA GPU SDK version 4.1; OpenMP programs were compiled using Visual Studio 2008; and all computation used double precision.

A. *Performance Speed up*

Ten classic SWO image filters were used as benchmarks for the evaluation – see Table 5. They were applied to a 3325×4765 image, with sliding windows of different sizes, including 3×3, 5×5, 7×7, 9×9. Figure 3 shows the performance above the baseline; for simplicity, it includes only the performance with 5×5 sliding windows.

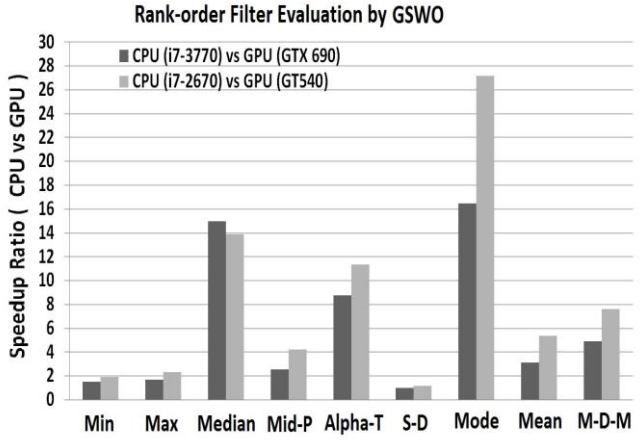


Figure 3. Speed-up performance evaluation of GSWO

On both platforms (a) and (b) described above – apart from the dilation and standard deviation filters, the speed-up ratios of the benchmarks are over one. Mean Filter and Mid-Point Filter are accelerated by GSWO up to 2-5 times.

The performance of the benchmarks with highly intensive computation (median filter, alpha-trimmed mean filter and mode filter) is particularly impressive, with speed-up ratios reaching up to 10-30. However, for image filters with low

computational demands, the speed-up ratios are also low. This is because the parallel regions in these filters represent only a small proportion of the entire running time. We have also evaluated the effect of sliding window size and computational complexity on the speed-up ratios. Figure 4 shows that when the sliding window size increases, the speed-up ratios of the GSWO-generated GPU code over the CPU baseline are significantly increased. Figure 4 also shows that the speed-up ratio for a filter increases according to its level of computational demand. GWSO accelerates those with the most intensive computation by up to 30×.

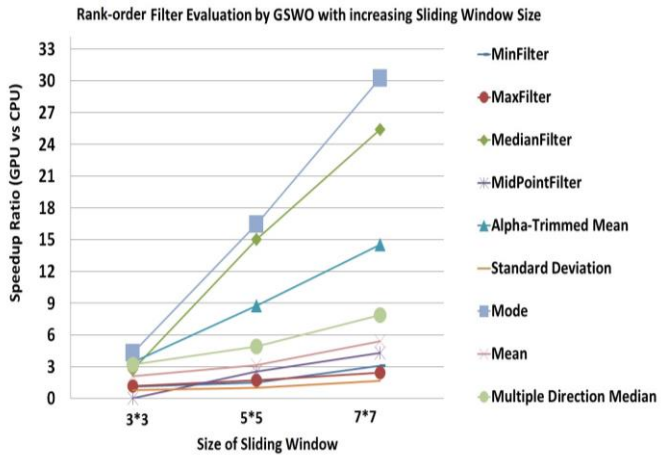


Figure 4. The impact of sliding window size on speed-up ratio

Figure 5 shows the correlation between the speed-up ratio and the kernel complexity for the 10 benchmarks using 5x5 sliding windows. The "**For**" and "**If**" statements were used to measure the complexity of the kernel – if there were two "**If**" statements within a "**For**" loop from 1 to 50, the kernel computation complexity was taken to be 50×2. The impact of different basic operations (arithmetic operations, assignments, tests, reads or writes) on numbers and types are ignored here. It can be seen that the benchmarks are clustered in the bottom-left and top-right corners of the diagram. For a given size of sliding window, the acceleration ratio increases noticeably as the complexity of the kernel grows.

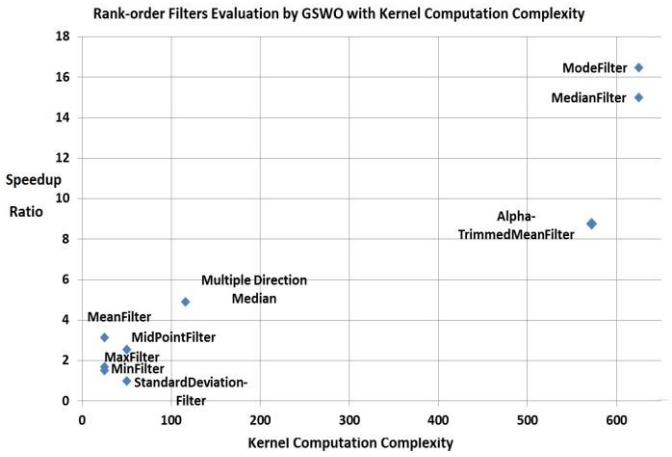


Figure 5. The impact of kernel computation complexity on speed-up

To conclude, the GSWO programming model is capable of accelerating the performance of most of the typical SWO image filters. It is particularly suitable for filters with highly intensive computations and large sliding windows. One possible bottleneck of the GSWO model is the limited size of on-chip memory on some GPUs, which thus may not fully support the application when a large sliding window and a complex filter is being used. However, GPU hardware is being given increased on-board memory on a regular basis and there is every prospect that this bottleneck is purely a temporary phenomenon.

B. *Acceleration Comparisons*

To compare the performance of GSWO to that of other CPU-to-GPU translators, we attempted to apply MINT [28], Bones [23], Par4All [27], OpenCV_GPU for CUDA [13], Polyhedral Benchmark [33], OpenACC PGI compiler [32] and OpenMP [39] to these SWO image filters. We found that:

- The original CPU code cannot be directly processed by most of the above tools. Bones and Par4All do not process C++, while Par4All has a limited capability in reduction operations and cannot produce GPU code for the maximum and minimum primitives.
- Polyhedral Benchmark is not a translator, but is simply a set of algorithms which can be used for testing the performance of translators. It cannot process the above applications. Also, Polyhedral and Bones are both algorithm skeleton based tools – the image filters under test are out of their scope.
- MINT was designed only for stencil operations and cannot handle SWO-based image filters. This was tested in our implementation. Also, the current version of MINT does not support C++.

Hence, the acceleration comparisons were mainly carried out using the OpenACC PGI compiler, OpenCV_GPU_Filter and OpenMP. We report test results over all types of benchmark. The SWO based image filters were implemented in C++ using Visual Studio 2008, based on external library OpenCV 2.4.3. The optimization flags used in VS2008 include Maximize Speed Optimization and Enable Intrinsic Functions. The test image had resolution 3325×4765, and 5×5, 9×9 and 11×11 sliding windows were used on hardware systems (b) and (c) described earlier. The quality of acceleration performance with other penalization tools was evaluated using speed-up ratio (CPU vs GPU), which measures the running time of the image filter part of the whole program. Figures 6(a),(b),(c) show the comparison results.

- The speedup performance of the OpenACC PGI compiler on all rank-order filters, erosion and dilation was compared with the GSWO model and the results with window size 9×9 are shown in Figure 6(a). For image filters with lower computation complexity (min, max, etc.), neither improve the performance as speedup ratios are lower than 1. However, for image filters with heavy computational complexity (median, Alpha-T, mode), both speed up the application by up to 10-26×. This implies that the OpenACC PGI compiler also follows the finding we demonstrated in Figure 5 – the acceleration ratio increases noticeably as the complexity of the kernels grows.

- Another noticeable issue is that the GSWO model has better acceleration performance than the OpenACC PGI compiler for the Median Filter, but is worse on Alpha-Trimmed Mean Filter and Mode Filter. This may because the GSWO model does not consider and use further optimization of the GPU kernel, but the OpenACC PGI compiler should optimize the GPU kernel in other ways, like using shared memory.

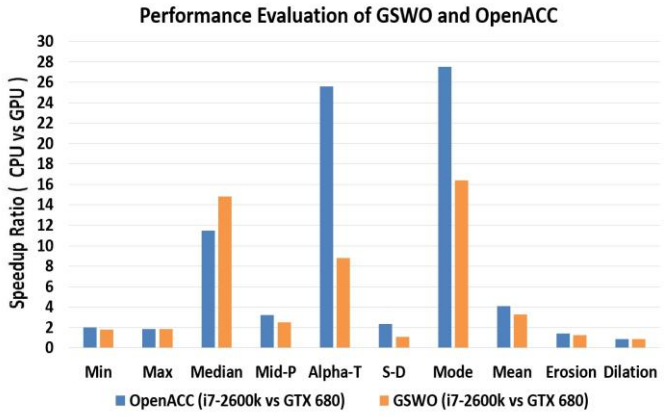


Figure 6(a). Speed-up ratio comparison between GSWO and OpenACC (9×9 sliding windows)

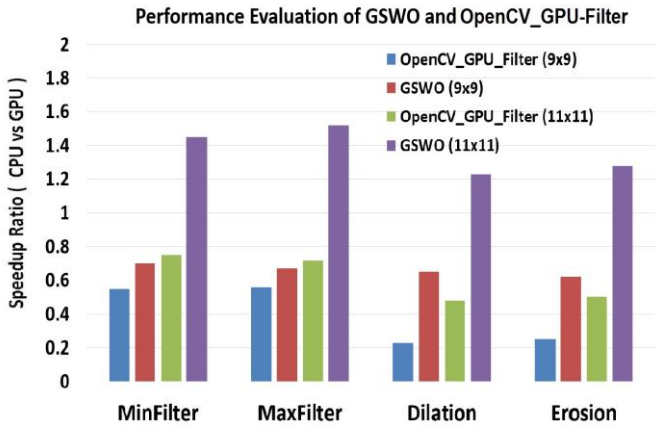


Figure 6(b). Speed-up ratio comparison between GSWO and OpenCV_GPU_Filter.

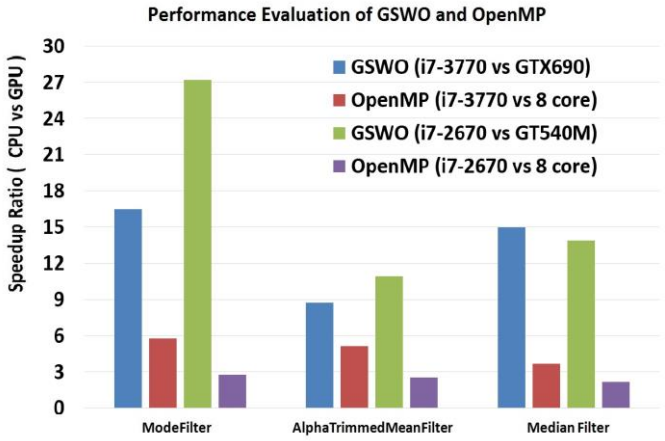


Figure 6(c). Speed-up ratio comparison between GSWO and OpenMP (5×5 sliding windows).

- The GPU module in the OpenCV library implements a number of GPU based image filter and algorithms. But it supports only a few types of image processing filter among our benchmarks in Table 6, which are max filter, min filter, erosion and dilation. We used the GPU filters function provided by OpenCV.2.4.3 to replace the relevant code part of image filters in our CPU implementations. The results of window size 9×9 and 11×11 are shown in Figure 6(b). It can be seen that GSWO has a consistently better performance than OpenCV_GPU_filter, but most of these filters run faster on a CPU implementation than on a GPU, probably because of their low computational complexity.

- We compared the performance of OpenMP when dealing with all rank-order filters, erosion and dilation with that of the GSWO model using 8 cores and 5×5 windows. We added OpenMP directives into the CPU filter part and enabled openmp2.0 language support from Visual Studio 2008. As with the OpenACC PGI compiler, the acceleration of OpenMP on filters with low computational complexity kernel is not noticeable, so the results in Figure 6(c) focus on the speedup performance of OpenMP and GSWO on filters with high computational complexity kernels. It shows that on modest hardware, GSWO can accelerate the benchmark filters up to 12-27×, which is higher than the performance of OpenMP on 8 cores with the speedup ratio up to 6-9×. It implies that the GSWO model has a competitive advantage over the existing OpenMP based tools.

In summary, for accelerating SWO image filter applications, the GSWO programming model is highly competitive with the state-of-the-art of automatic CPU-to-GPU source translators, which makes it attractive in comparison to other research focused tools, such as hiCUDA, MINT, Par4All and Bones. Also, the GSWO model has improved acceleration ability than traditional OpenCV_GPU_filters or OpenMP supports. While compared to commercial products such as PGI, the GSWO model usually has a lower acceleration performance, but it still can exceed PGI compiler on median filters.

Another advantage of GSWO over the PGI compiler is that the output of GSWO is CUDA code, which is readable, and more importantly, revisable. The source-to-source conversion provides users with the opportunity to carry out further modification of the converted source code according to their needs, as well as to use the machine-generated code as an example to support their learning of GPU programming techniques.

C. *Case Study 1: Camera Fingerprint Measurement*

Camera fingerprint measurement is a particularly popular topic in information forensics and security. In most approaches, denoising methods are applied to a set of images that are known to come from a given camera [34-35]. In this section, we evaluate GSWO using sample camera fingerprint measurement applications from an industrial source, IME [36].

The C++ code applies a 3×3 median filter for image denoising, subsequently measuring the camera fingerprint by comparing the denoised image with the original image. The number of images was 39, each of resolution 3648×2736. When applying

the median filter, the images were split into a number of Regions of Interest (ROIs). The loop in this function can be parallelized.

A key feature of the application is that, in the workflow, its algorithm skeleton contains loops outside the sliding window operation, as shown in Figure 7. Most existing CPU-to-GPU translators cannot be successfully applied to these codes because of the complex algorithm skeletons. MINT [28] can indirectly process the sample code by revising the original function into three separate sub-functions and generating three kernels. However, the GPU performance is even slower than the CPU performance because repeated CUDA memory allocation functions are called within the main loops. In contrast, the GSWO model is fully capable of processing the code. The results are shown in Table 8. The CPU version is a sequential version without the use of OpenMP.

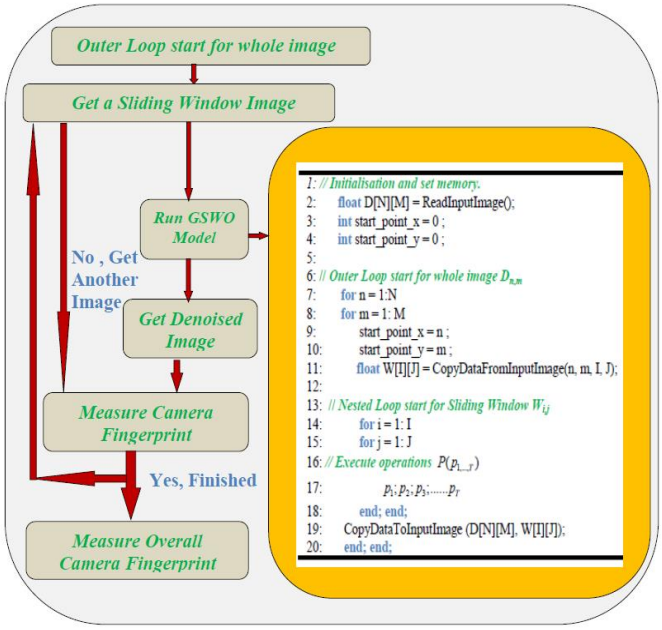


Figure. 7. Camera finger measurement application from IME

Table 8 shows that GSWO speeds up the whole application performance by up to 3-4×, on average, while maintaining the same accuracy as the original CPU source code. While we are able to achieve a 6× performance gain in the kernel region, the overheads associated with use of the GPUs (e.g. data transfer between CPU and GPUs) reduce the performance advantage somewhat. However, the overall performance gain is still satisfactory. Notably, most of the existing CPU-to-GPU tools (e.g. Bones [23], Par4All [27], Polyhedral Benchmark [33], PGI [32]) cannot process the source code of this application.

A noticeable result in Table 8 is that the speedup ratio will reduce if you deploy a mid-performance graphic card alongside a high-speed CPU. The first reason is that we used the running time of the whole program to calculate the ratio, which adversely affects the speed ratio. The second reason is that we considered a real application that contains many C++ source and head files. In order to use the GSWO model in the application, users have to break up the data dependency of the functions in the C++ file and also need to add some extra CPU code to transfer the image data format so as to be acceptable by the GSWO model. This process adds some running time to CPU programs, and further reduces the speed-up ratio. However, it does not mean that the GSWO model is not useful since the running time of the whole program is eventually reduced.

Table 8. Evaluation of Camera Fingerprint Measurement

	CPU (s)	*GPU (s)*	*Speedup*	*Details*
GTX 690	28	8.2	3.4	Whole Application (1 image)
GT 540	84	19	4.3	Whole Application (1 image)
GTX 690	16	3.9	4.2	Kernel Region (1 image)
GT 540	51	8.3	6.3	Kernel Region (1 image)
GTX 690	1118	280	3.8	Whole Application (39 image)
GT 540	3047	707	4.3	Whole Application (39 image)
GTX 690	663	153	4.3	Kernel Region (39 image)
GT 540	2023	325	6.2	Kernel Region (39 image)

D. *Case Study 2: Document Segmentation*

Large-scale document digitization is another research issue with potential application in museums and libraries. The performance of an OCR system depends heavily on document layout analysis, region segmentation and text-line segmentation, which is a time-consuming procedure for large-scale and high-resolution document digitization.

We applied SWO-based dilations and erosions to process sample newspaper document images from IMPACT [37], which is one of the most widely recognized large-scale document digitization projects of recent years. The processed newspaper images, of resolution 3595×5194, were then evaluated by a region segmentation method [38]. The results are shown in Table 9 and in Figure 8 in Appendix B.

Table 9. Document Analysis code evaluation by GSWO

System (a) (seconds)			System (b) (seconds)			
CPU	**GPU**	**Speed up**	**CPU**	**GPU**	**Speed up**	**Details**
0.26	1.0	0.3	1.3	1.6	0.8	3 × 3 dilation
1.19	1.2	0.9	4.3	1.9	2.2	5 × 5 dilation
3.69	1.2	2.9	18.2	2.3	7.9	9 × 9 dilation
0.24	1.03	0.2	1.5	2.3	0.7	3 × 3 erosion
1.10	1.0	1.1	5.7	2.1	2.7	5 × 5 erosion
3.35	1.1	2.9	16.8	2.5	6.6	9 × 9 erosion

When the dilation or erosion operator uses 5×5 sub-windows, the GSWO translator speeds up the performance by a factor of 1-3×. When the dilation or erosion operator uses sub-windows smaller than 5×5, the GPU performance becomes slower than the CPU performance due to the overheads mentioned earlier. This result confirms that the GSWO programming model is most suitable for applications with high kernel complexity.

Figure 8 illustrates the evaluation results of the region segmentation method [38], which extracts text regions from newspaper images, based on a hybrid of erosion and dilation. In

the original newspaper image, a large number of text regions are missed due to the low density of the characters. When 3×3 dilation operators were used, most of text regions were segmented but two pieces of the text regions in the middle of the document were still missing. When 5×5 dilation operators were used, the output quality improved but one text region in the middle of the document was still missing. By use of 9×9 dilation operators, all text regions in the newspaper were successfully segmented. Table 8 shows that GSWO is capable of speeding up the application performance by up to 6×.

E. *Usability Comparisons*

To compare the practical usability of the GSWO model to that of other CPU-to-GPU translators, we provided CPU-to-GPU translators listed in Section IV.B to non-expert GPU users for accelerating their real applications. The initial findings are as below.

- The majority of research tools, like MINT [28], Bones [23], Par4All [27], etc. cannot process C++ and are hard to learn. A common phenomenon is that these tools can attain very high speed-up ratios using particular forms of optimization specifically tuned to the task, but such performance can rarely be achieved in real-world, practical applications.
- OpenMP and OpenCV based GPU filters are the easiest approaches for non-expert GPU users. The GPU filter functions in OpenCV are nearly as same as its CPU functions. The use of OpenMP in CPU programs only needs to put one directive into the SWO filter kernel. However, their acceleration performance is not as good as GSWO.
- OpenACC PGI compiler is the strongest CPU-to-GPU tool in the market. Both of the OpenACC PGI compiler and the GSWO model are directive based CPU-to-GPU translators. The usability comparison between them is reported.

For usability, comparison was made between the GSWO model and the PGI compiler regarding ease of use and ease of learning. The evaluation involved four parts, including understanding of loop patterns and pragmas, use of pragmas, the effect of CPU code revision and debug diagnostics. Feedback was collected, via a questionnaire, from non-expert GPU users in four GPSME project partners (IME, AnSmart, B3C, RotaSoft) based on use of the GSWO model and the OpenACC PGI compiler. The results are shown in Table 10.

Table 10. Learning and use by inexperienced GPU users

	GSWO	OpenACC
Understand loop pattern	fair	fair
Understand Basic pragma	good	fair
Understand Memory Management pragma	fair	moderate
Understand Kernel Generation pragma	good	moderate
Number of total pragmas used	8-14	5-10
Number of Memory Management pragmas used	5-9	1
Number of Kernel Generation pragmas used	4	3-5
CPU code revision	Moderate	Moderate
Extra lines of code	Moderate	Moderate
Running sufficiently fast	Good	Good
Debug diagonistics	Yes	Yes
Readabliltiy of output code	Yes	No
Overall rate of usability	Good	Good

Table 10 demonstrates that the overall usability of both tools is rated as good by new users, though each has advantages and disadvantages. The OpenACC compiler supports the learning and use of memory management pragmas rather better than GSWO, but GSWO performs better for kernel generation pragmas. This is because the kernel generation design of the GSWO model focuses particularly on SWO image filter applications, whereas the OpenACC compiler aims at more generic cases.

The memory management pragmas of the GSWO model are extensions of MINT to support more flexible data transfer. The OpenACC is a well-known standard with mature design on memory management. These issues lead to fewer pragmas being used in OpenACC for memory management than in the GSWO model, but a variable number of pragmas being used in kernel generation. The CPU code revision and debug diagnostics are both required in OpenACC and GSWO in the parallelization of SWOs applications. However, GSWO has an considerable advantage concerning the readability of the CUDA code output, which can significantly help new users to track errors and potentially improve performance.

V. Discussion and Limitations

From the results above, we conclude that the GSWO programming model is capable of accelerating the performance of many SWO-based image applications. By applying the GSWO model, we are able to achieve significant performance gains in sliding window operations, particularly those that are computationally demanding. Compared to many existing automatic CPU-to-GPU programming models, the GSWO model has an enhanced usability and acceleration performance. While the GSWO model has no significant advantages on acceleration and usability over the OpenACC PGI compiler, it still demonstrates a possibility of using an easy-to-use CPU-to-GPU code translator to accelerate the SWOs based Image Filter applications with good performance.

A. *Flexibility Usability Tradeoff*

Actually, when designing a directive based automatic CPU-to-GPU source translator, there is a tradeoff between the flexibility and the easy-to-use of pragmas. It is true that increasing the number of clauses may give more flexible management of device memory or optimization and lead to better acceleration performance, but it also increases the difficulty of use. The pragma design in the GSWO model tries to strike a balance between flexibility and ease of use. Compared to the OpenACC PGI compiler, GSWO less simple in broader generic cases, but is very simple to use in SWO applications. It is important for users to know the performance difference to decide whether to use a specialized tool and sacrifice flexibility or use a more general tool with slower results.

B. *Memory Transfer Overhead*

A key feature of performance evaluation on GSWO model is to face the real image processing applications but not the pure kernel codes. The acceleration performance of the whole image

processing applications is more sensible than speedup on kernel codes to many industrial users. So the execution time of these experiments will include the costs on loading applications on CPU, transferring data between CPU and GPU, and running the kernel in GPU. As shown in Table.8, if we only consider the kernel region acceleration but not the whole application, the speedup performance on GTX540 will increase from 4.29 to 6.17. It means that the memory transfer overhead indeed could affect the overall performance.

C. *Use of Shared Memory*

The optimization of the CUDA kernel code in the GSWO is a potential extension work in future. So far, the key idea of kernel generation in the GSWO model is based on the parallelization of a typical operational procedure of an SWO image filter. The procedure of kernel generation is an SWO procedure based translation, but is not a strictly sentence by sentence translation. The codes presenting operations within a sliding window are eventually transferred into CUDA kernel body. Optimizing CUDA kernel code with shared memory has been considered in this paper. We have chosen three main filters from Figure 5 (Median filter, Mode filter and Alpha-Trimmed Mean Filter) to implement a hand-revised CUDA kernel optimization by using shared memory to store temp data in a sliding window. It shows that the overall performance increased by 20-25% compared to when we do not use shared memory. So it implies that the uses of shared memory in GSWO model will potentially speedup the over all performance. But due to the diverse code structure of image operations transferring to kernel code, it will take much time to implement the automatic code translation with shared memory in GSWO model based translator for more generic image operations. How to optimize the GSWO model with shared memory for more generic image operations will be investigated in future work.

D. *Other issues*

There are a few minor limitations on GSWO. Firstly, it is not suitable to operations that are not parallelizable or are relatively light in computational terms. But sufficient numbers of parallelizable, computationally demanding tasks exist in practice for GSWO to make a significant contribution. Secondly, a possible bottleneck of the GSWO model is the limited size of on-chip memory available on some GPU devices; this may not fully support the use of a large sliding window and an intensive-computation filter. However, the regular expansion in on-board GPU memory will certainly help to mitigate this limitation. Thirdly, the memory management pragmas of the GSWO model are not simple for a non-expert to understand and use correctly, though they can be successful with a little care.

We believe that the benefits of the GWSO approach greatly outweigh these disadvantages, and that GWSO affords a new and effective way of accelerating SWO image processing applications.

VI. Conclusion and Future Work

This paper has presented an annotation-based programming model, GSWO, which supports sliding window operations in a wide range of image filters. It enables users to carry out source-to-source conversion of self-implemented image filters from CPU to GPU in a highly automated manner. Compared to many existing automatic CPU-to-GPU programming models, the GSWO model has an enhanced usability and acceleration performance. The experimental results show its good speed-up and usability in a variety of image processing applications, at a similar level to the state-of-the-art tool OpenACC PGI compiler.

Future work will introduce new pragmas to extend the GSWO model for more general time-consuming image processing applications, such as object detection. Meanwhile, it expects to be compatible with existing research tools [28] [31] [33] to optimize the GPU performance of this tool – shared memory optimization, loop aggregation and register optimization. The support of OpenCL output is also under consideration.

References

1. Y. H. Chen, S. J. Horng, R. S. Run, J. L. Lai, R. J. Chen, W.C. Chen, Y. Pan and T. Takao, "A scan-based configurable,programmable, and scalable architecture for sliding window-based operations," *IEEE Trans. Computers*, vol.48, no. 6, pp.615-627, June. 1999.
2. P. Shivakumara, G. H. Kumar, D. S. Guru, and P. Nagabhushan, "Sliding window based approach for document image mosaicing," *Journal of Image and Vision Computing* , vol. 24, issue 1, pp. 94–100, 2006.
3. Y.R. Wang, W.H. Lin and S.J. Horng: "A sliding window technique for efficient license plate localization based on discrete wavelet transform," *Expert Syst. App,* vol. 38, issue 4, pp. 3142-3146, April. 2011.
4. X. Xu, E.L. Miller, D.Chen and M. Sarhadi, "Adaptive two-pass rank order filter to remove impulse noise in highly corrupted images," *IEEE Trans. Image Processing*, vol 13, no. 2, pp. 238-247, Feb. 2004.
5. Y. Nie and K.E. Barner, "Fuzzy Rank LUM Filters," *IEEE Trans. Image Processing*, vol 15, no. 12, pp. 3636-3654, Dec. 2006.
6. P. Soille and H. Talbot, "Directional morphological filtering," *IEEE Trans. Pattern Analysis and Machine Learning*, vol 23, issue 11, pp. 1213-1329, Nov. 2001.
7. J. Gil and R. Kimmel, "Efficient dilation, erosion, opening and closing algorithms," *IEEE Trans. Pattern Analysis and Machine Learning*, vol 24, issue 12, pp. 1606-1617, Dec, 2002.
8. M. J. Thurley and V. Danell, "Fast morphological image processing open-source extensions for GPU processing with CUDA," *IEEE Journal of Selected Topics in Signal Processing*, vol 6, no 7, pp. 849-856, Nov. 2012.
9. C. H. Wu and S. J. Horng, "Fast and scalable selection algorithms with applications to median filtering," *IEEE Trans. Parallel and Distributed Systems*, vol 14, no. 10, pp. 983 – 992, Oct. 2003.
10. M. M. Bronstein, "Lazy sliding window implementation of the bilateral filter on parallel architectures," *IEEE Trans. Image Processing*, vol 20, no. 6, pp.1751-1757, June. 2011.
11. P. N. Happ, R. Q. Feitosa, C. Bentes and R. Farias, "A region-growing segmentation algorithm for GPUs," *IEEE Geoscience and Remote Sensing Letters*, vol.10, no.6, pp.1612-1617, Nov. 2013.
12. W.J. Dally, "The GPU Computing Era," *IEEE Micro.* vol 30, issue 2, pp.56-69, April. 2010.
13. OpenCV_gpu. (Dec, 2013), "Opencv_2.4.8.," Available [Online]: http://docs.opencv.org/modules/gpu/doc/introduction.html

14. GpuCV. (Oct, 2010), “Gpucv: GPU-accelerated Computer Vision,” Available [Online]: http://picoforge.int-evry.fr/cgi-bin/twiki/view/Gpucv/Web/
15. A. Reiner. (Sep, 2010), “Etothepi CUDA-Image-Processing”, Available [Online]: https://github.com/etotheipi/CUDA-Image-Processing
16. M. G. Sanchez, V. Vidal, J. Bataller and J. Arnal, “A parallel method for impulsive image noise removal on hybrid CPU/GPU systems”, *Procedia Computer Science*, vol 18, pp.2504-2507, June, 2013.
17. GPSME. (Oct, 2013), “A General Toolkit for “GPUtilisation” in SME Applications”, Available [Online]: www.gp-sme.co.uk
18. R. C. Hardie and C. Boncelet, “LUM filters: a class of rank-order-based filters for smoothing and sharpening”, *IEEE Trans. Signal Processing*, vol 41, issue 3, pp.1061-1076, Mar. 1993.
19. D. B. Min, J. B. Lu and M. N. Do, “Depth video enhancement based on weighted mode filtering”, *IEEE Trans. Image Processing*, vol 21, no. 3, pp.1176-1190, Mar. 2012.
20. R. Oten, D. Figueiredo and J. P. Rui, “Adaptive alpha-trimmed mean filters under deviations from assumed noise model”, *IEEE Trans. Image Processing*, vol 13, no. 5, pp.627-639, Mar. 2004.
21. Y.M.Y. Hasan and L. J. Karam, “Morphological text extraction from images”, *IEEE Trans. Image Processing*, vol 9, no. 11, pp. 1978-1983, Nov. 2000.
22. S. Chen and R. M. Haralick, “Recursive erosion, dilation, opening and closing transforms”, *IEEE Trans. Image Processing*, vol 4, no. 3, pp. 335-345, March. 1995.
23. C. Nugteren and H. Corporaal. “Introducing ‘Bones’: A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons.” In GPGPU-5: 5th Workshop on General Purpose Processing on Graphics Processing Units. ACM, 2012.
24. J. Enmyren and C. K. Kessler. “SkePU: A multi-backend skeleton programming library for multi-GPU systems”, *In Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications* (HLPP-2010), Baltimore, Maryland, USA. ACM, pp. 5-14, 2010.
25. M.M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev and P. Sadayappan., “A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs”, *Proc. Int’l Conf. Supercomputing*, NewYork, USA. ACM, pp. 225-234, 2008.
26. A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, “A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction”, *in Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (GPGPU ’10), New York, NY, USA, ACM, pp. 51–61, 2010.
27. HPC Project, (Oct, 2011), “Par4all automatic parallelization,” Available [Online]: http://www.par4all.org.
28. D. Unat, X. Cai, and S. B. Baden. ” Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C”, *In ICS ’11: International Conference on Supercomputing*, New York, NY, USA, ACM, pp. 214-224, 2011.
29. S.Z., Ueng, M. Lathara, S.S. Baghsorkhi and W. W. Hwu, “CUDA-lite: Reducing GPU Programming Complexity “, *Proc. Int’l Workshop Languages and Compilers for Parallel Computing*, Berlin, Heidelberg. Springer, pp. 1-15. 2008.
30. T. Han and T. Abdelrahman, “hiCUDA: High-Level GPGPU Programming”, *IEEE Trans. Parallel and Distributed Systems*, vol 22, no. 1, pp. 78-90, Jan. 2011.
31. The OpenACC Standard, The OpenACC™ Application Programming Interface http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, November 2011.
32. The Portland Group, (June. 2009), “PGI Fortran and C Accelerator Programming Model “, Available [Online]: http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.0.pdf
33. L.-N. Pouchet. (Nov. 2011), “PolyBench: The Polyhedral Benchmark Suite.” Available [Online]: http://www.cse.ohio-state.edu/~pouchet/software/polybench/
34. G. J. Bloy, “Blind camera fingerprinting and image clustering,” *IEEE Trans. Pattern Anal. Mach. Intell*., vol. 30, no. 3, pp. 532–534, Mar, 2008.
35. V. Conotter and G. Boato, "Analysis of sensor fingerprint for source camera identification," *Electronic Letters*, vol. 47, no. 25, pp. 1366-1367, Dec. 2011.
36. IME. (Nov. 2012), Image Forgery Detection, Ltd. Available [Online]: http://www.imagemetry.com/
37. IMPACT. (Dec. 2011),”Improving Access to Text, EU FP7 project”, http://www.impact-project.eu
38. P. Yang, A. Antonacopoulos, C. Clausner, S. Pletschacher, “Grid-based modelling and correction of arbitrarily warped historical document images for large-scale digitization”, in: *Proceedings of the 2011 Workshop on Historical Document Imaging and Processing, HIP '11*, ACM, Beijing, China, pp. 106–111, Sep. 2011.
39. U. Bondhugula, M. Baskaran, S. Krishnamoorthy and J. Ramanujam, A. Rountev, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer, *ACM SIGPLAN Programming Languages Design and Implementation (PLDI)*, Tucson, Arizona. Jun 2008.
40. Pluto, (Dec, 2013), “A polyhedral automatic parallelizer and locality optimizer for multicores”, Available [Online]: http://pluto-compiler.sourceforge.net

Appendix A

The system structure and translation flow of the C2GPU toolkit is illustrated in Figure 9. The input to the toolkit is C/C++ source code annotated with pragmas. Once the source file is read, the ROSE frontend constructs the AST tree and passes it to the core of the C2GPU toolkit. The core of the toolkit traverses the AST and queries the parallel regions. Directives in a parallel region go through the components of *Identifier*, *Analyser* and *Optimizer* in the toolkit core. The *Translator* component uses the rules from the above components to transform the AST. The output from the toolkit is CUDA or OpenCL source code generated by unparsing the transformed AST. This paper uses only the CUDA code generated by the toolkit.

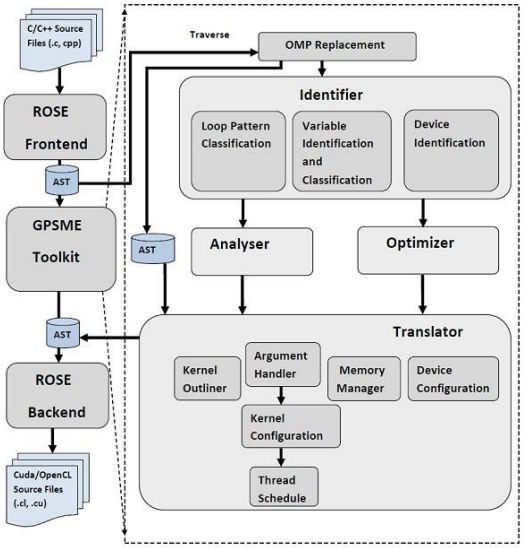
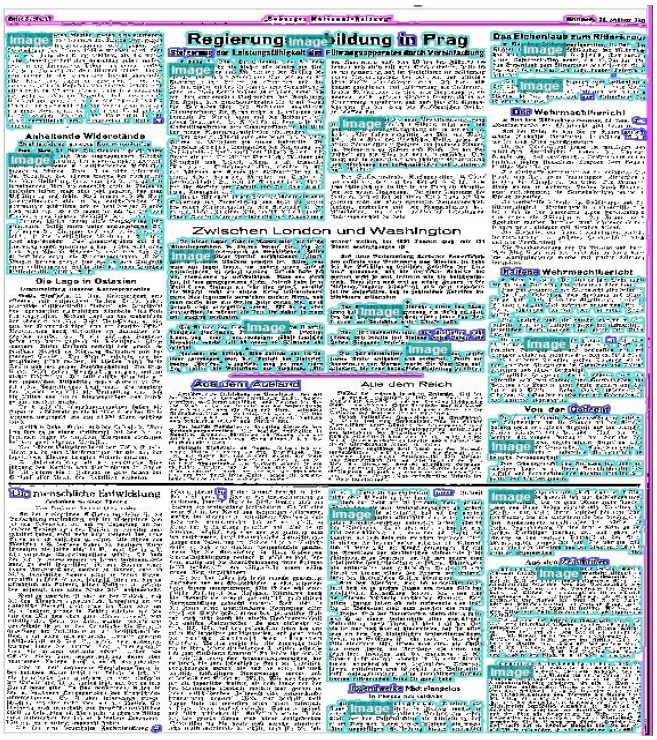
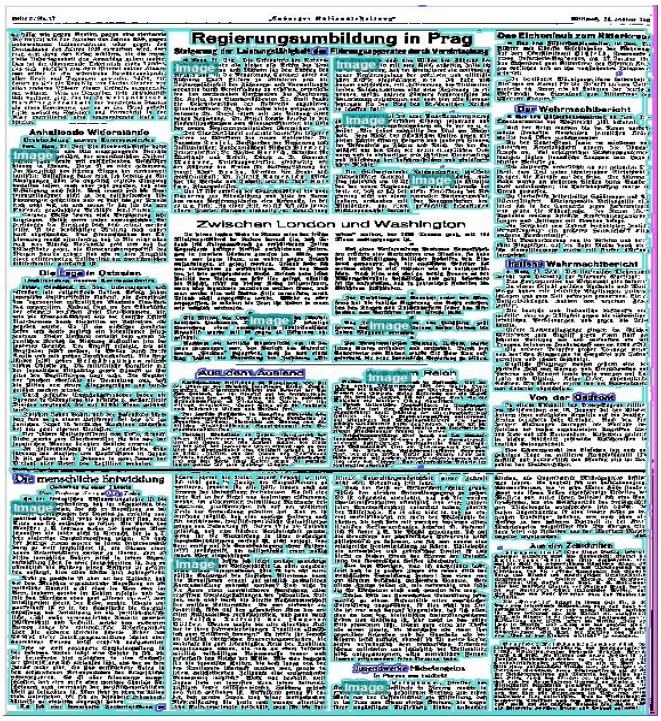


Figure 9 GSWO Core Library

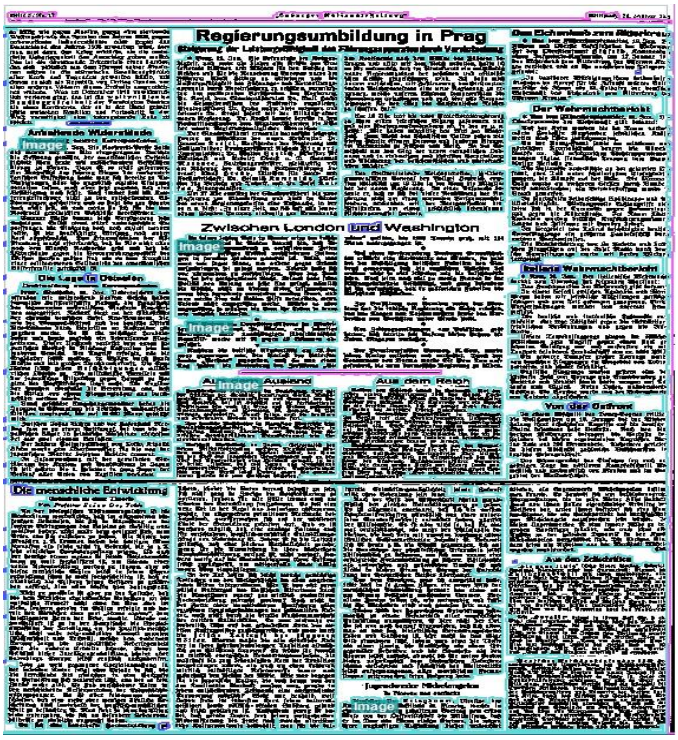
Appendix B



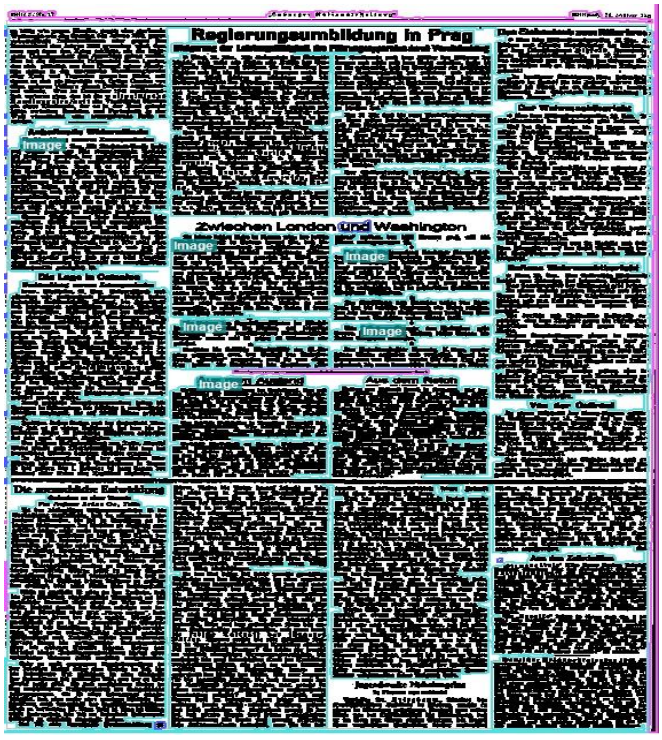
(a) Original



(b) 3×3 dilation



(c) 5×5 dilation



(d) 9×9 dilation

Figure 8. Region segmentation results of GSWO-produced images.

Appendix C

Table 9. CUDA code of memory creation pragmas

Directives	*Descriptions*

copyByTexture (D, toDevice, N, M, Bind, char)	cudaChannelFormatDesc desc_1;
	desc_1 = cudaCreateChannelDesc<unsigned char>();
	tex_dev_3_D.normalized = false;
	tex_dev_3_D.addressMode[0] = cudaAddressModeClamp;
	tex_dev_3_D.addressMode[1] = cudaAddressModeClamp;
	tex_dev_3_D.filterMode = cudaFilterModePoint;
	cudaMallocArray(&array_dev_3_D,&desc_1,N,M);
	cudaMemcpyToArray(array_dev_3_D,0,0,((char *)D),sizeof(char) * N * M, cudaMemcpyHostToDevice);
	cudaBindTextureToArray(tex_dev_3_D, array_dev_3_D);

copyMalloc1DArray (W, toDevice, I, J,pitch1)	cudaMallocPitch(((void **)(&d_dev_5_W)),&::pitch123,I * sizeof(float), J);
copyMalloc1DArray (im_GPU, toDevice, I, J, pitch1, InKernel)	cudaMallocPitch(((void **)(&d_dev_6_im_GPU)),&::pitch123,I * sizeof(float), J);

copy2DArrayTo1DArray (W, toHost, I, J, W_1D, 2DTo1D)	int i_1;
	int j_1;
	for (i_1 = 0; i_1 < J; ++i_1)
	for (j_1 = 0; j_1 < I; ++j_1) {
	W_1D[i_1*I+j_1] = W[i_1][j_1]; }

copyMemcopy2D (W, HosttoDevice, I, J, pitch1,W_1D)	cudaMemcpy2D(d_dev_5_W,sizeof(float) * I, W_1D, pitch123, sizeof(float) * I, J, cudaMemcpyHostToDevice);
copyMemcopy2DToArray (W, DevicetoDevice, ROI_w, ROI_h, pitch1)	cudaMemcpy2DToArray(array_dev_4_W,0,0, d_dev_5_W,pitch123,sizeof(float) * ROI_w, ROI_h, cudaMemcpyDeviceToDevice);

copyBindTexture (W, DevicetoDevice, W, float, Bind)	cudaChannelFormatDesc desc_3;
	desc_3 = cudaCreateChannelDesc<float>();
	cudaBindTextureToArray(&tex_dev_4_W, array_dev_4_W,&desc_3);

copyMemcopy2D (im_GPU, DevicetoHost, I, J, pitch1, W_1D)	cudaMemcpy2D(W_1D,sizeof(float) * I, d_dev_6_im_GPU,pitch123,sizeof(float) * I, J, cudaMemcpyDeviceToHost);

copy2DArrayTo1DArray (W_1D, toHost, I, J, W, 1DTo2D)	int i_2;
	int j_2;
	for (i_2 = 0; i_2 < J; ++i_2)
	for (j_2 = 0; j_2 < I; ++j_2) {
	W[i_2][j_2] =W_1D[i_2*I+j_2]; }